# Hybrid Fuzzing for Structured Inputs: Integrating Grammar-Aware and Mutation-Based Techniques

**Bachir Bendrissou**

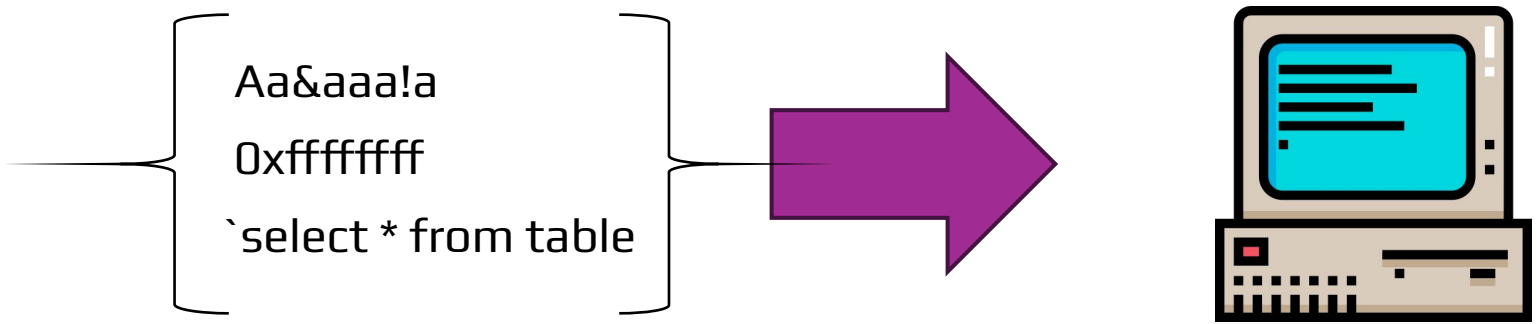
*Supervised by:* **Cristian Cadar, Alastair Donaldson**

*Imperial College London*

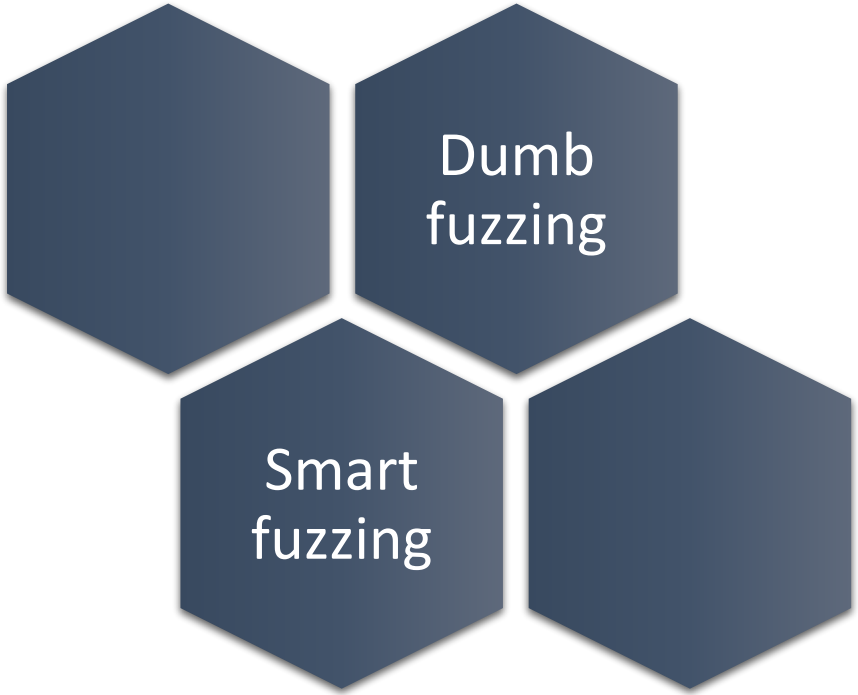
# Outline

1. Grammar Mutation for Testing Input Parsers (TOSEM)
2. Structure Resilience in Greybox Fuzzing via Automated Error Recovery (ASE DS)

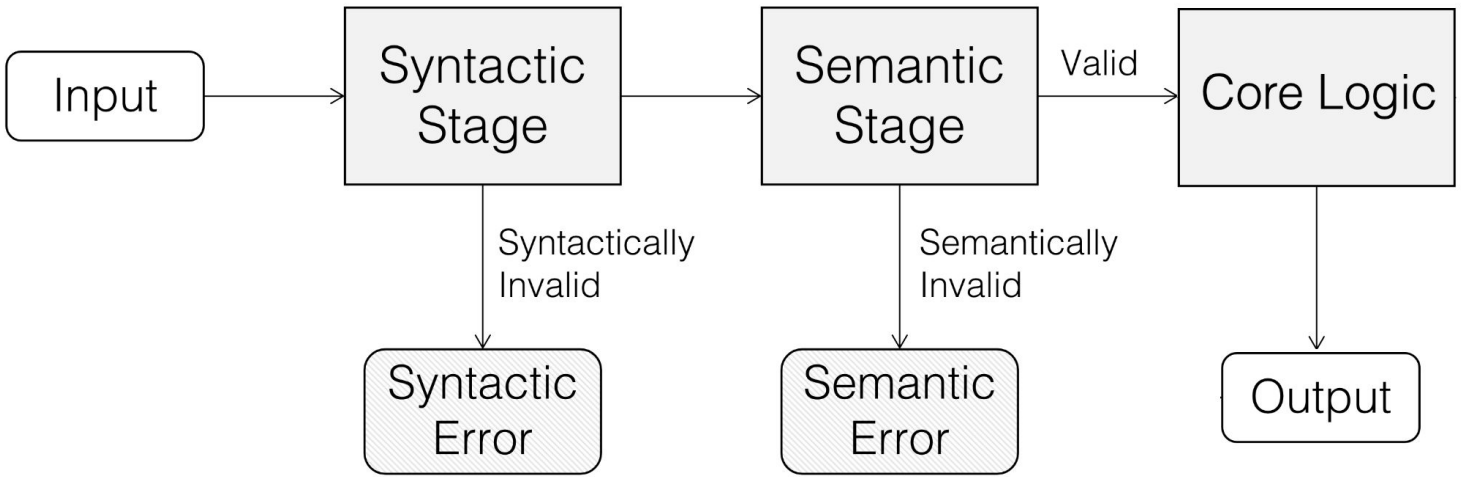
# *Fuzz Testing (Fuzzing)*



# *Main Approaches*



# *Input Processing Pipeline*



# Grammar-based fuzzing: **smart**

Generates **valid** inputs

# Grammar-based fuzzing: **smart**

Generates **valid** inputs

Great! Valid inputs go **deep**

# Grammar-based fuzzing: **smart**?

Generates **valid** inputs

Great! Valid inputs go **deep**

But…

# Grammar-based fuzzing: **smart**?

Generates **valid** inputs

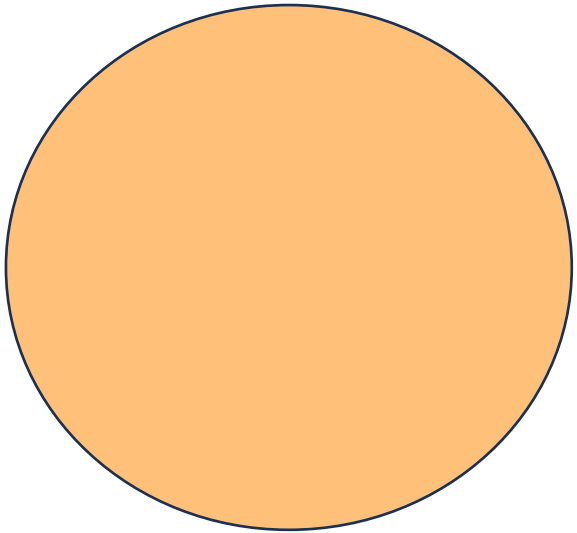
Great! Valid inputs go **deep**

But…

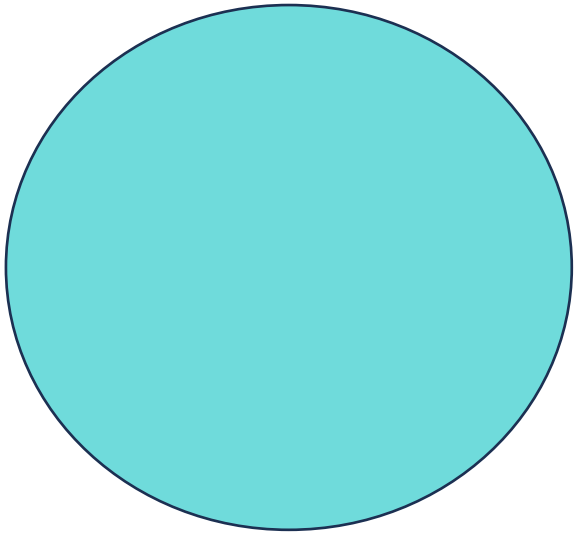
- What if SUT does not conform to grammar?
- What about subtle bugs triggered by **almost**-valid inputs?

# Grammars vs. SUTs

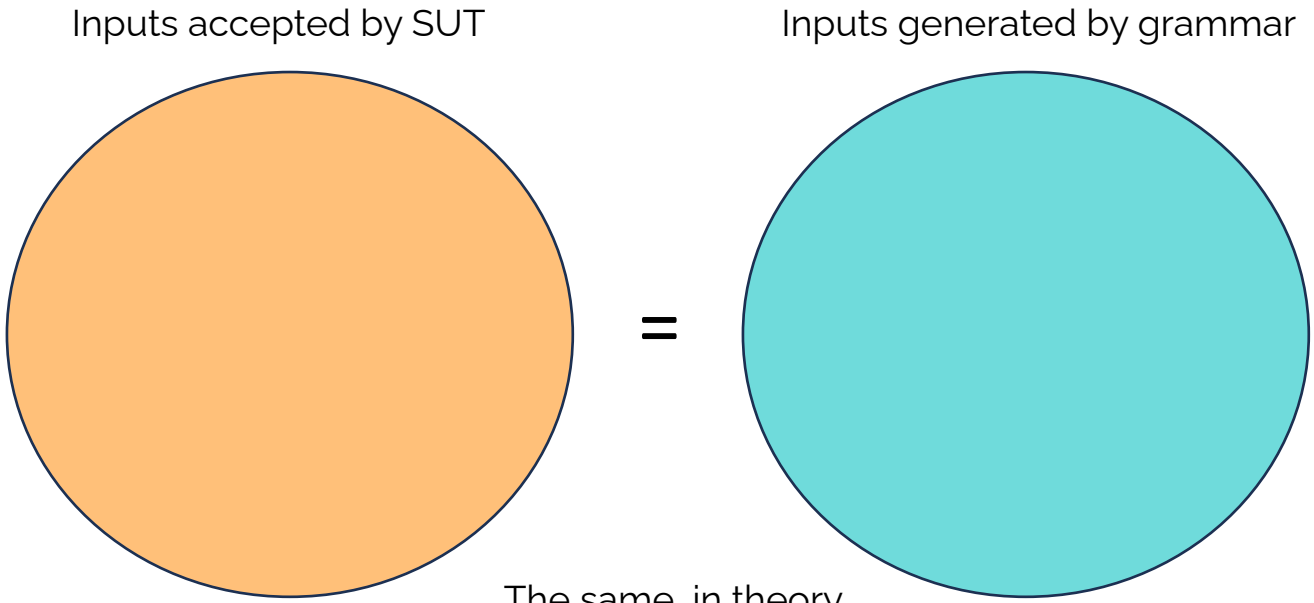
Inputs accepted by SUT



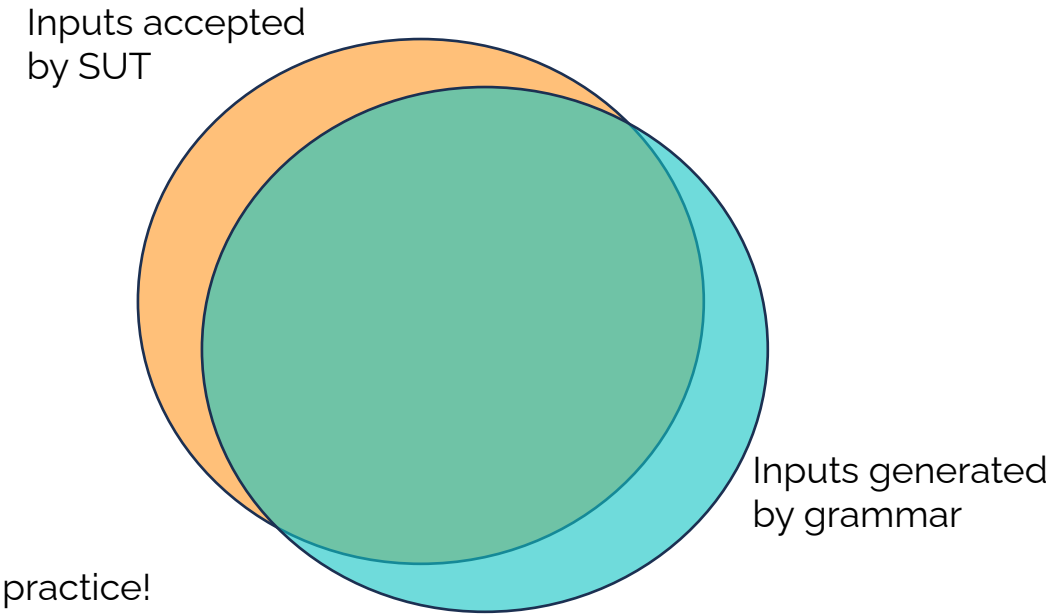
Inputs generated by grammar



# Grammars vs. SUTs

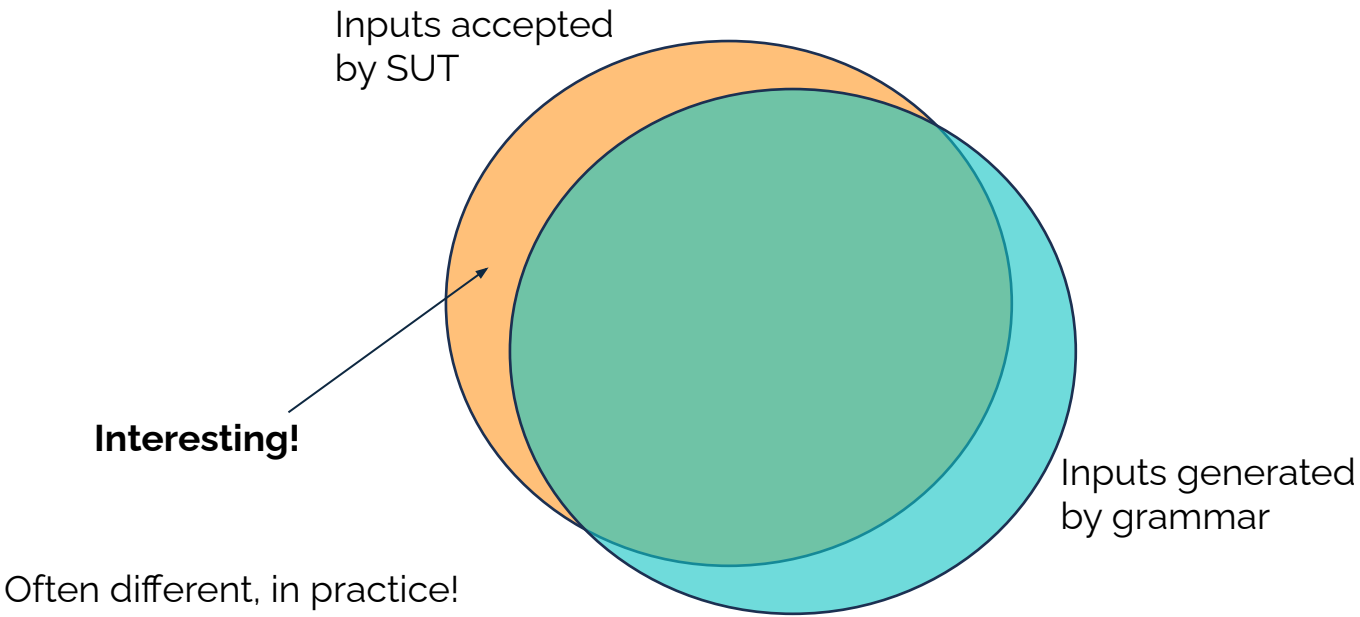


# Grammars vs. SUTs




Often different, in practice!

# Grammars vs. SUTs

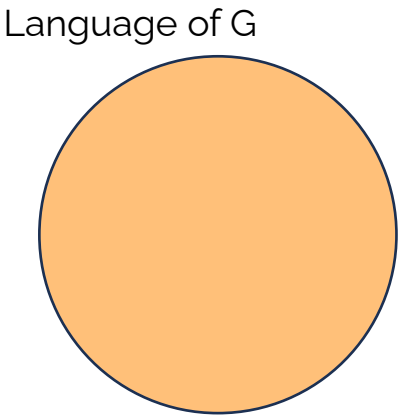



Often different, in practice!

# Our idea: **grammar mutation**

**Given:**

- grammar G for input format
- SUT that claims to consume input format

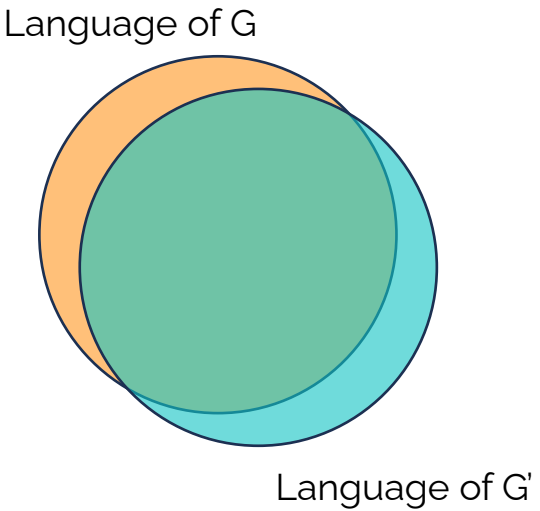


# Our idea: **grammar mutation**

**Given:**

- grammar G for input format
- SUT that claims to consume input format

**Mutate** G to get **mutant grammar** G’



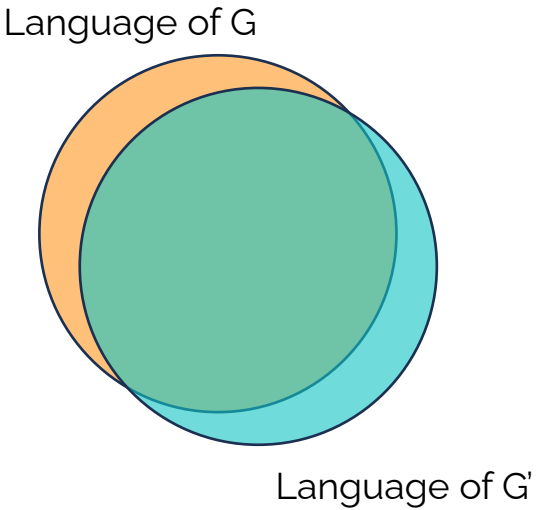
# Our idea: **grammar mutation**

**Given:**

- grammar G for input format
- SUT that claims to consume input format

**Mutate** G to get **mutant grammar** G’

**Fuzz** SUT using G’



# Our idea: **grammar mutation**

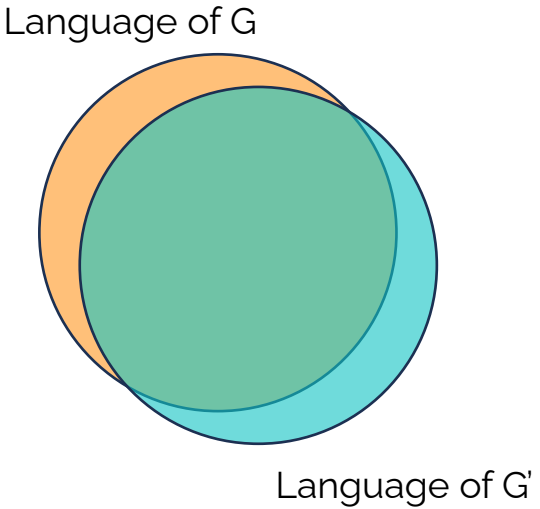
**Given:**

- grammar G for input format
- SUT that claims to consume input format

**Mutate** G to get **mutant grammar** G’

**Fuzz** SUT using G’

**Identify** non-G inputs accepted by SUT



# Our idea: **grammar mutation**

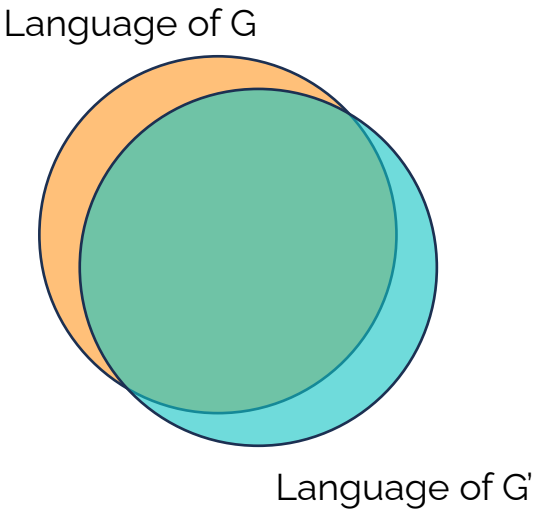
**Given:**

- grammar G for input format
- SUT that claims to consume input format

**Mutate** G to get **mutant grammar** G’

**Fuzz** SUT using G’

**Identify** non-G inputs accepted by SUT




Try many different mutant grammars at random

# **Example:** mutating JSON

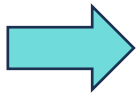
```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

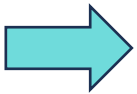
```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```



```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

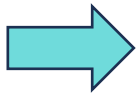
```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```



```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

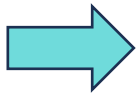
```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```



```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value*)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

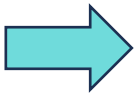
```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```



```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value*)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

# **Example:** mutating JSON

```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```



```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value*)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' (STRING | HEX) HEX HEX HEX ;
```

# Empirical Evaluation

Input formats

**JSON**

**Lua**

**URL**

**XML**

# Empirical Evaluation

Input formats

**JSON**

**Lua**

**URL**

**XML**

SUTs

cJSON
Parson
simdjson

luac
LuaJIT
py-lua-parser

aria2
curl
wget

fast-xml-parser
libxml2
pugixml

# Empirical Evaluation

Input formats

**JSON**

**Lua**

**URL**

**XML**

SUTs

cJSON
Parson
simdjson

luac
LuaJIT
py-lua-parser

aria2
curl
wget

fast-xml-parser
libxml2
pugixml

Tools:

- **Grammarinator**
- **Gmutator**
- **G+M** -> Grammarinator + string mutation

24 hour runs

3 repeat runs per configuration

# Evaluation Criteria

1. **Discrepancy Bugs:** Identify parsing issues
    a. Accept-Invalid
    b. Reject-Valid
2. **Code Coverage:** Unique code of lines covered by Gmutator and G+M

Table 3. Unique `accept-invalid` and ANTLR issues discovered by GMUTATOR and G+M.

| # | SUT | Description | Status | # |
|---|---|---|---|---|
| 1 | PY-LUA-PARSER | Fail to reject unassigned global variable | Fixed | 9 |
| 2 | PY-LUA-PARSER | Fail to reject an invalid escape sequence | Fixed | |
| 3 | PY-LUA-PARSER | Parsing standalone name tokens | Fixed | |
| 4 | PY-LUA-PARSER | Missing function call arguments | Fixed | |
| 5 | PY-LUA-PARSER | Fail to parse chained comparisons | Fixed | |
| 6 | WGET | Semicolon incorrectly handled in userinfo | Fixed | |
| 7 | ANTLR | Parsing LUA long comment as short comment | Fixed | |
| 8 | ANTLR | Ambiguity in URL grammar | Fixed | |
| 9 | ANTLR | Underscore not allowed in XML NameStartChar | Fixed | |
| 10 | CJSON | Fail to reject invalid escape character | Confirmed | 4 |
| 11 | FAST-XML-PARSER | Fail to reject multiple root nodes | Confirmed | |
| 12 | FAST-XML-PARSER | Validation of invalid XML declarations | Confirmed | |
| 13 | FAST-XML-PARSER | Parsing an invalid XML element | Confirmed | |
| 14 | PARSON | Accepting invalid array | Rejected | 2 |
| 15 | PARSON | EOF not enforced | Rejected | |
| 16 | CJSON | Accepting invalid integers | Reported | 2 |
| 17 | PY-LUA-PARSER | Literal string gets parsed as LUA code | Reported | |

# Discrepancy Bugs

**Discrepancies** found for both SUTs:

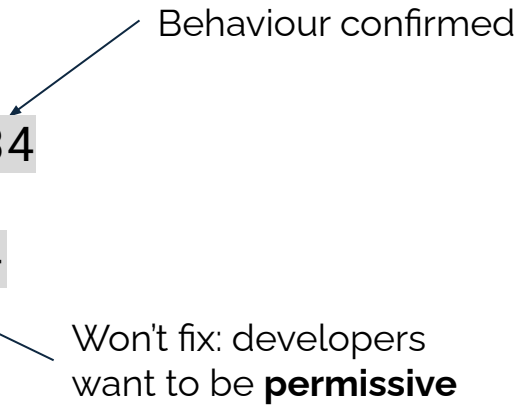
- cJSON accepts invalid unicode: `\uZ234`
- Parson accepts invalid JSON: `{} {}`

# Discrepancy Bugs

**Discrepancies** found for both SUTs:

- cJSON accepts invalid unicode: `\uZ234`

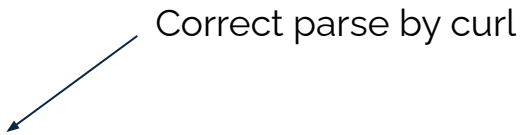
- Parson accepts invalid JSON: `{} {}`



# Discrepancy Bug: CVE-2024-38428

**Discrepancy** found in URL SUTs:

curl "**`http://a;bc@xyz`**"
curl: (6) Could not resolve host: xyz



# Discrepancy Bug: CVE-2024-38428

**Discrepancy** found in URL SUTs:

curl "**`http://a;bc@xyz`**"
curl: (6) Could not resolve host: xyz

Correct parse by curl

wget "**`http://a;bc@xyz`**"
wget: unable to resolve host address ‘a;bc@xyz’

Userinfo incorrectly parsed as hostname

Table 4. Differential line coverage between G+M and GMUTATOR. The highest value for each SUT is in bold.

| Tool | cJSON | PARSON | SIMDJSON | LUAC | LUAJIT | PY-LUA | ARIA2 | CURL | WGET | FAST-XML | LIBXML2 | PUGIXML |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **GMUTATOR** | **10** | **4** | **116** | **79** | **131** | 7 | 0 | 5 | 3 | 1 | **183** | **8** |
| **G+M** | 6 | 1 | 33 | 20 | 22 | **16** | **58** | **192** | **9** | **16** | 88 | 2 |

# Takeaway

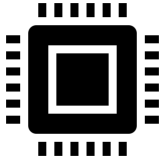
1. Both Gamutator and G+M are able to find discrepancy bugs
2. Gmutator overall finds more unique code coverage
3. Both tools complement each other

# Structure Resilience in Greybox Fuzzing via Automated Error Recovery

**Bachir Bendrissou**

*Imperial College London*

*Supervised by:* **Cristian Cadar, Alastair Donaldson**

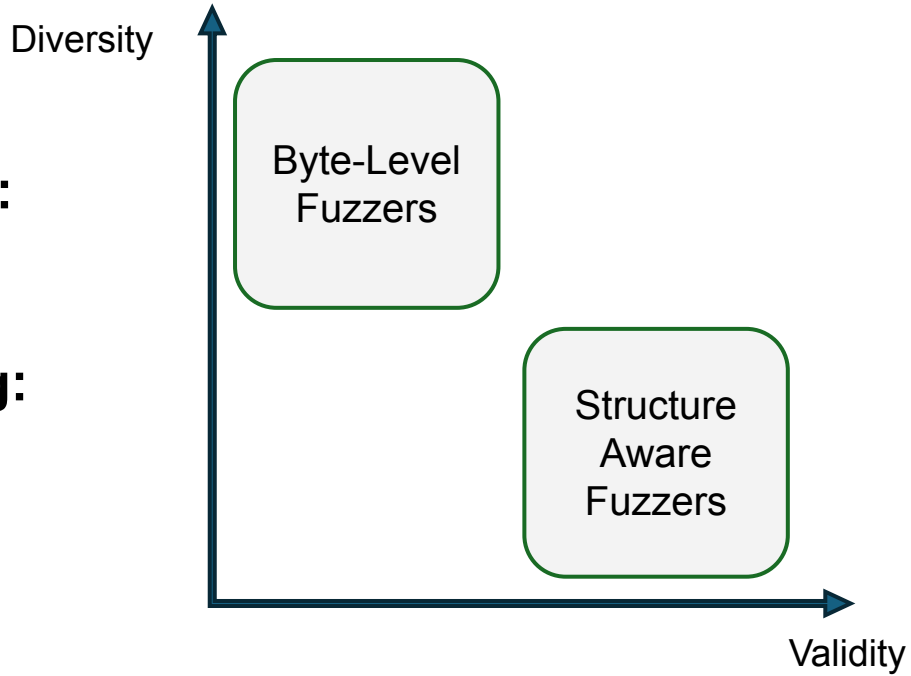


# *State of the Art*

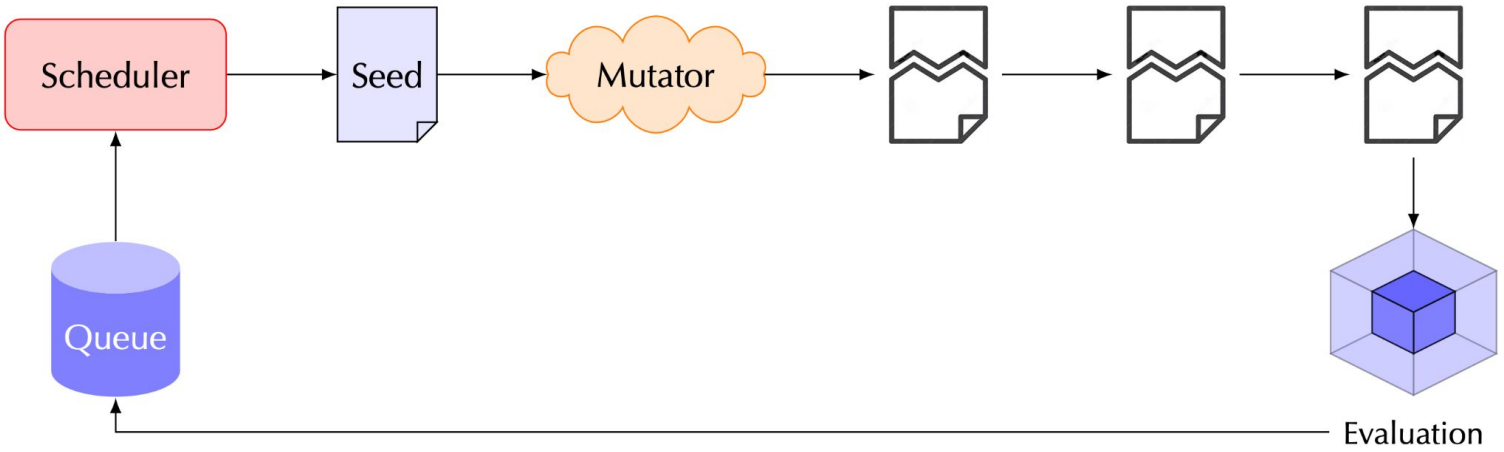
1. **Mutation-based fuzzing:**

   diverse tests, but invalid
2. **Grammar-based fuzzing:**

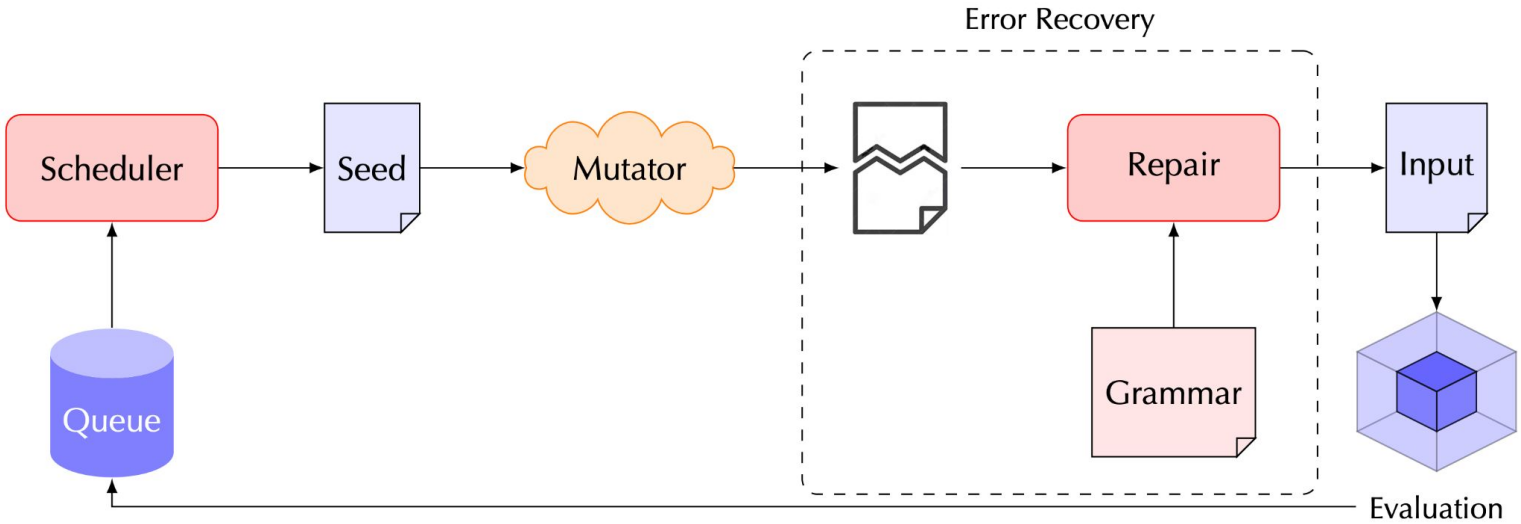
   valid tests, but uniform



# State of the Art: American Fuzzy Lop (AFL)



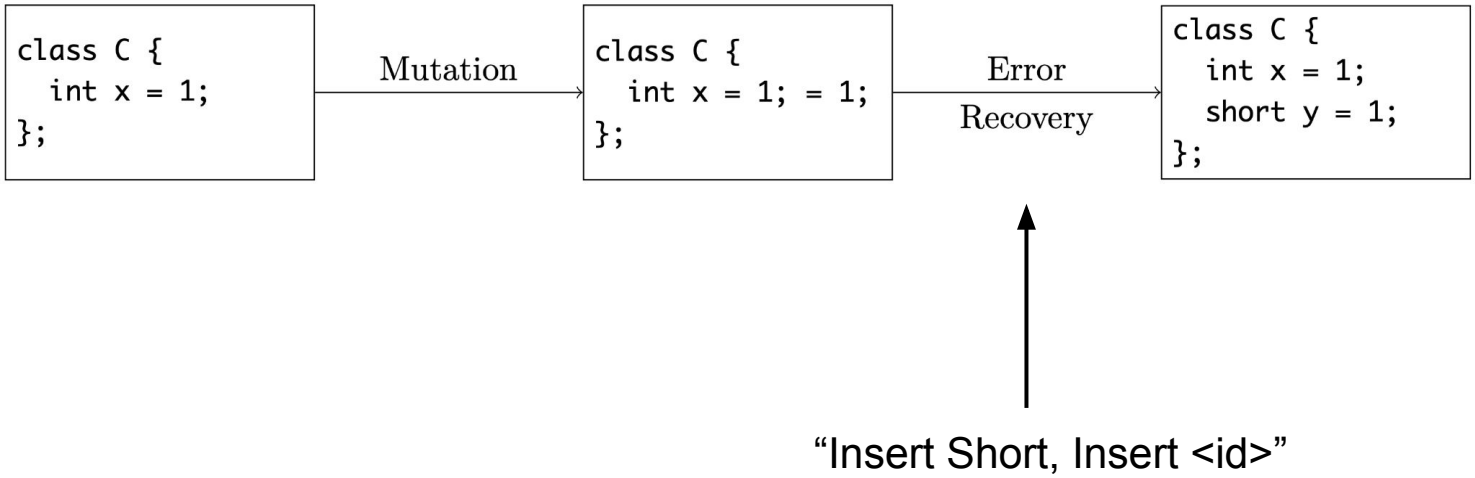
# *Proposed Solution: AFLRepair*



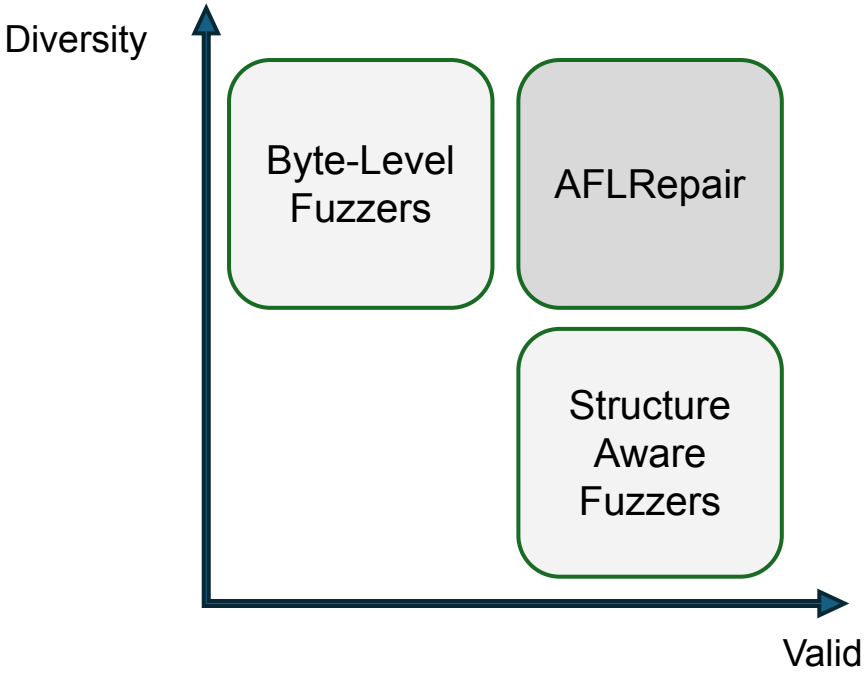
# *Automatic Error Recovery*

1. Originally used to report programming errors and suggest fixes
2. Search-based approach
3. Highly precise and efficient

# Automatic Error Recovery



# ★ *The Differentiating Factor*




Produce highly diverse test cases while maintaining structure validity

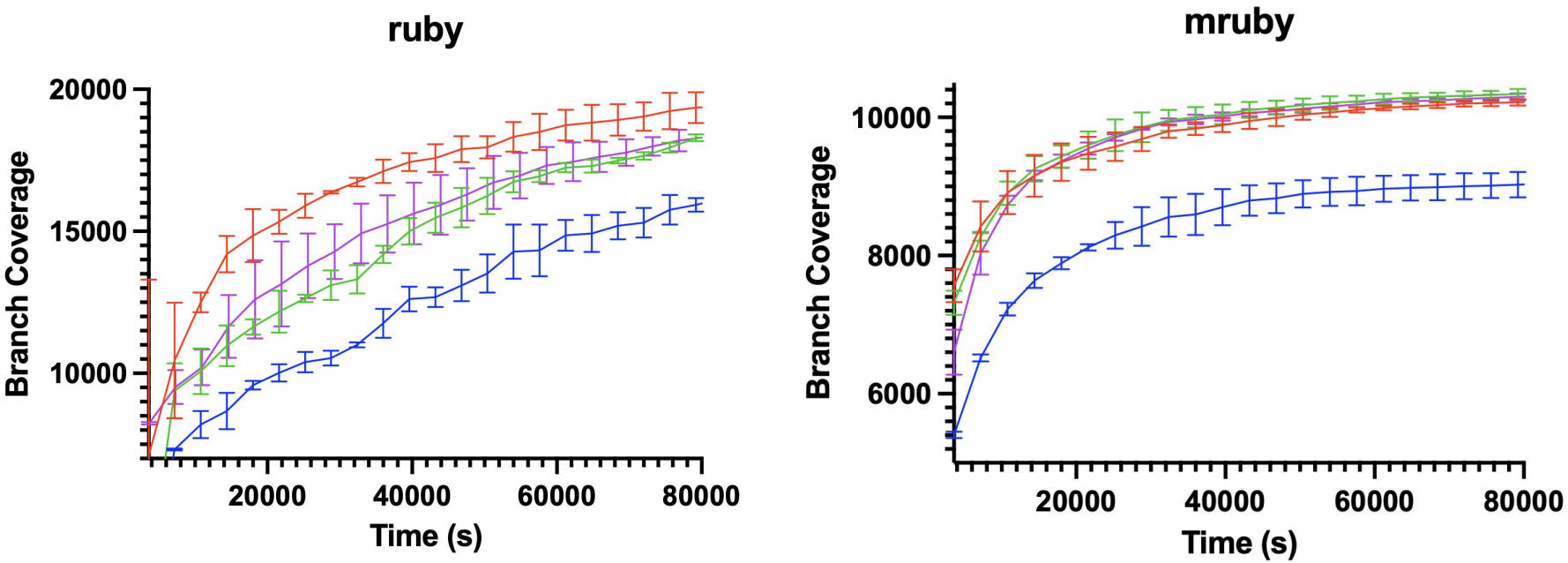
# *Evaluation Plan: Systems Under Test*

| SUT | Language | LOC |
|---|---|---|
| Lua | Lua | 15 k |
| LuaJIT | lua | 50 k |
| Ruby | Ruby | 768k |
| mRuby | Ruby | 73k |
| V8 | JavaScript | 1.6M |
| ChakraCore | JavaScript | 559k |
| PHP | PHP | 888k |

# *Evaluation Plan: Tools*

1. **Nautilus:** Generates and mutates inputs using a grammar
2. **AFL Standard:** Imports Nautilus inputs as seed corpus, and performs byte-level mutations
3. **AFLRepair (full):** Imports Nautilus inputs as seed corpus, performs byte-level mutations, and repairs every mutated input
4. **AFLRepair (partial):** Same as AFLRepair (full), but only repairs mutated inputs 50% of the time

# *Results: Code Coverage*



# *Results: Bugs Found*

| Program | Language | # |
|---|---|---|
| Lua | Lua | 1 |
| LuaJIT | lua | 1 |
| Ruby | Ruby | 2 |
| mRuby | Ruby | 1 |

# *Conclusion*

- Grammar-based fuzzing is **scalable and effective**
- Grammar Mutation adds **edge-case** inputs
- Validity alone isn't enough — **diversity** matters
- AFLRepair achieves both: valid + diverse inputs
- Builds on recent advances in fuzzing and parsing

# Thank you

## Grammar Mutation for Testing Input Parsers

BACHIR BENDRISSOU, Imperial College London, United Kingdom
CRISTIAN CADAR, Imperial College London, United Kingdom
ALASTAIR F. DONALDSON, Imperial College London, United Kingdom

Grammar-based fuzzing is an effective method for testing programs that consume structured inputs, particularly input parsers. However, if the available grammar does not accurately represent the input format, or if the system under test (SUT) does not conform strictly to the grammar, there may be an impedance mismatch between inputs generated via grammars and inputs accepted by the SUT. Even if the SUT *has* been designed to strictly conform to the grammar, the SUT parser may exhibit vulnerabilities that would only be triggered by slightly invalid inputs. Grammar-based generation, by construction, will not yield such edge case inputs. To overcome these limitations, we present two mutational-based approaches: Gmutator and G+M. Both approaches are built upon Grammarinator, a grammar-based generator. Gmutator applies mutations to the grammar input of Grammarinator, while G+M directly applies byte-level mutations to Grammarinator-generated inputs. To evaluate the effectiveness of these techniques (Grammarinator, Gmutator, G+M) in testing programs that parse various input formats, we conducted an experimental evaluation over four different input formats and twelve SUTs (three per input format). Our findings suggest that both Gmutator and G+M excel in generating edge case inputs, facilitating the detection of disparities between input specifications and parser implementations.

## Syntactic Resilience in Greybox Fuzzing: Automated Error Recovery

Bachir Bendrissou
Imperial College London
London, United Kingdom
b.bendrissou@imperial.ac.uk

**Abstract**

*Fuzz testing*, an automated technique that introduces random data inputs to systems, has demonstrated remarkable effectiveness in identifying vulnerabilities. Its scalability and automation have made it a focal point of interest in both academic and industrial settings. However, traditional fuzzing techniques often struggle to generate diverse, rare inputs that conform to a program's input specifications, thereby limiting their full potential. To address these challenges, I propose AFLRepair, a novel approach that applies random mutations to program inputs and subsequently repairs the syntax of any resulting invalid inputs. AFLRepair leverages byte-level mutations to create a wide array of test cases while ensuring their validity, facilitating the exploration of diverse execution

is a software bug in OpenSSL cryptography software library, compromised thousands of web servers worldwide. Similar bugs have been found in other critical software such as kernels and compilers.

Several automated software testing techniques have been developed to detect faults and improve the reliability of software programs. Fuzz testing (fuzzing) [8, 11] has gained significant traction in recent years. A fuzzer in its basic form works by generating random inputs, executing a target program on the generated inputs, while observing the behaviour of program. A greybox fuzzer normally employs code coverage feedback to guide its execution towards interesting program paths. A fuzzer may either generate inputs from a predefined template or by mutating existing inputs.

The AFL fuzzer [12] is a popular greybox mutation-based fuzzing